Table of Contents

[Collections 1](#_Toc462230585)

[*Concurrency* 2](#_Toc462230586)

Collections*: A collection — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data.*

Collections Framework

1. ***Interfaces****: These are abstract data types that represent collections*
2. ***Implementations****: These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.*
3. ***Algorithms****: These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.*

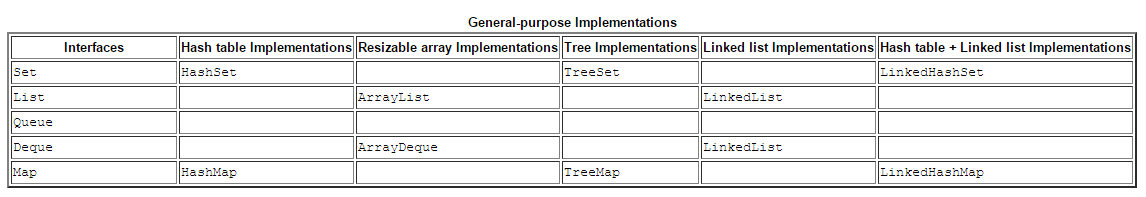
Benefits

1. *Reduces programming effort*
2. *Increases program speed and quality*
3. *Allows interoperability among unrelated APIs*
4. *Reduces effort to learn and to use new APIs*
5. *Reduces effort to design new APIs*
6. *Fosters software reuse*

Interfaces



Implementations



Algorithms

* [Sorting](https://docs.oracle.com/javase/tutorial/collections/algorithms/index.html#sorting)
* [Shuffling](https://docs.oracle.com/javase/tutorial/collections/algorithms/index.html#shuffling)
* [Routine Data Manipulation](https://docs.oracle.com/javase/tutorial/collections/algorithms/index.html#rdm)
* [Searching](https://docs.oracle.com/javase/tutorial/collections/algorithms/index.html#searching)
* [Composition](https://docs.oracle.com/javase/tutorial/collections/algorithms/index.html#composition)
* [Finding Extreme Values](https://docs.oracle.com/javase/tutorial/collections/algorithms/index.html#fev)

# Concurrency

Process - A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own.

Thread - Threads are sometimes called *lightweight processes*. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Sleep(), join(), interrupt()

## Synchronization

This form of communication is extremely efficient, but makes two kinds of errors possible: *thread interference* and *memory consistency errors*. The tool needed to prevent these errors is *synchronization*.

* [Thread Interference](https://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html) describes how errors are introduced when multiple threads access shared data.
* [Memory Consistency Errors](https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html) describes errors that result from inconsistent views of shared memory.
* [Synchronized Methods](https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html) describes a simple idiom that can effectively prevent thread interference and memory consistency errors.
* [Implicit Locks and Synchronization](https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html) describes a more general synchronization idiom, and describes how synchronization is based on implicit locks.
* [Atomic Access](https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html) talks about the general idea of operations that can't be interfered with by other threads.

Deadlock, Livelock, starvation

**High Level Concurrency Objects**

**Lock** A lock is a tool for controlling access to a shared resource by multiple threads

**Executors** define a high-level API for launching and managing threads. Separate thread management and creation from the rest of the application

**Concurrent collections** make it easier to manage large collections of data, and can greatly reduce the need for synchronization.

**Atomic variables** have features that minimize synchronization and help avoid memory consistency errors.

**ThreadLocalRandom** (in JDK 7) provides efficient generation of pseudorandom numbers from multiple threads

Executor Interfaces

The Executor interface provides a single method, execute, designed to be a drop-in replacement for a common thread-creation idiom. If r is a Runnable object, and e is an Executor object you can replace

The ExecutorService interface supplements execute with a similar, but more versatile submit method. Like execute, submit accepts Runnable objects, but also accepts Callable objects, which allow the task to return a value. The submit method returns a Future object, which is used to retrieve the Callable return value and to manage the status of both Callable and Runnable tasks.

The ScheduledExecutorService interface supplements the methods of its parent ExecutorService with schedule, which executes a Runnable or Callable task after a specified delay. In addition, the interface defines scheduleAtFixedRate and scheduleWithFixedDelay, which executes specified tasks repeatedly, at defined intervals.